Creative list styling

- When to use a list
- List types
  - Unordered list (ul)
  - Ordered list (ol)
  - Description list (dl)
- Simple list styling
- ::marker
  - Custom counters
  - Limitations of ::marker styling
- Styling lists that don't look like lists
- Custom markers with ::before
- List attributes
  - Reversed lists
  - Split lists
- Multi-column list layout
- Custom properties
- Description list styling
- Resources

What comes to mind when you think of a list? The most obvious example is a shopping list—the most simple of lists, a collection of items in no particular order. But we use lists in all sorts of ways on the web. A collection of upcoming concerts at a venue? Very likely a list. A multi-step booking process? Quite possibly a list. A gallery of images? Even that could be considered a list of images with captions.

In this article we'll dive into the different HTML list types available to us on the web and when to use them, including some attributes you might not be familiar with. We'll also take a look at some useful and creative ways to style them with CSS.

When to use a list

An HTML list element should be used when items need to be semantically grouped. Assistive technologies (such as screen readers) will notify the user that there is a list, and the number of items. If you think about, say, a grid of products on a shopping site, knowing this information would be very helpful. Therefore, using a list element might be a good choice.

List types

When it comes to markup, we have a choice of three different list elements available to us:

- Unordered list
- Ordered list
- Description list

Which one to pick depends on the use case.

Unordered list (ul)

The unordered list element (`<ul>`) is most useful when the items in the list don't correspond to any particular order. By default this will display as a bulleted list. An example is a shopping list, where the order doesn't matter.

{% Img src="image/kheDArv5csY6rvQUJDbWRscckLr1/59pGr3TkE9oBx4QDCiUM.png", alt="A shopping list of items such as bread, milk, apples.", width="800", height="387" %}

A more common example on the web is a navigation menu. When building a menu, it is good practice to wrap the `ul` in a `nav` element and to identify the menu with a label, to aid assistive technologies. We should also identify the current page in the menu, which we can do using the `aria-current` attribute:

```
<nav aria-label="Main">
  <ul>
    <li>
      <a href="/page-1" aria-current="page">Menu item 1</a>
    </li>
    <li>
      <a href="/page-2">Menu item 2</a>
    </li>
    <li>
```

```
      <a href="/page-2">Menu item 2</a>
    </li>
      …
    </ul>
</nav>
```

This article on menu structure outlines a number of recommendations to ensure our navigation menus are accessible to all.

Ordered list (ol)

An ordered list element (`<ol>`) is the best choice when the order of items is important, such as a multi-step process. By default, list items are numbered. An example could be a set of instructions, where steps must be completed in order.

{% Img src="image/kheDArv5csY6rvQUJDbWRscckLr1/MtcTt0l6WlxVFhZyVvaN.png", alt="A list detailing the steps required to make tea with milk.", width="800", height="620" %}

{% Codepen { user: 'web-dot-dev', id: 'mdxjLEd', height: 500, theme: 'light', tab: 'result' } %}

Both `<ol>` and `<ul>` elements may only contain `<li>` elements as their direct children.

Description list (dl)

A description list contains terms (`<dt>` elements) and descriptions (`<dd>`). Each term can be accompanied by more than one description. Possible use cases could include a glossary of terms, or perhaps a restaurant menu. Description lists are not displayed with any markers by default, although browsers tend to indent the `<dd>` element.

In HTML it is permitted to group terms with their accompanying descriptions using a `<div>`. This can be useful for styling purposes, as we'll see later on.

```
<!-- This is valid -->
<dl>
    <dt>Term 1</dt>
    <dd>This is the first description of the first term in the list</dd>
    <dd>This is the second description of the first term in the list</dd>
    <dt>Term 2</dt>
    <dd>This is the description of the second term in the list</dd>
</dl>
```

```
<!-- This is also valid -->
<dl>
    <div>
        <dt>Term 1</dt>
        <dd>This is the first description of the first term in the list</dd>
        <dd>This is the second description of the first term in the list</dd>
    </div>
    <div>
        <dt>Term 2</dt>
        <dd>This is the description of the second term in the list</dd>
    </div>
</dl>
```

Simple list styling

One of the simplest uses of a list is within a block of body text. Quite often these simple lists don't need elaborate styling, but we might want to customize the markers of an ordered or unordered list to some extent, such as with a brand color, or by using a custom image for our bullets. We can do quite a lot with `list-style` and the `::marker` pseudo-element!

{% Aside %} For a refresher on the basics of list styling, check out the Learn CSS tutorial, as well as a deep-dive into using `::marker` in this custom bullets guide. Read on for some things you might not already know! {% endAside %}

::marker

In addition to giving our list markers some basic styling, we can create cyclical bullets. Here we're using three different image URLs for the `content` value of the `::marker` pseudo-element, which adds to the hand-written feel of our shopping list example (as opposed to just using a single image for all):

```
::marker {
    content: url("/marker-1.svg") ' ';
}

li:nth-child(3n)::marker {
    content: url("/marker-2.svg") ' ';
```

```
}

li:nth-child(3n - 1)::marker {
    content: url("/marker-3.svg") ' ';
}
```

{% Codepen { user: 'web-dot-dev', id: 'poLZVbr', height: 500, theme: 'light', tab: 'result' } %}

Custom counters

For some ordered lists we might want to use the counter value, but append another value to it. We can use the `list-item` counter as a value for our marker's `content` property and append any other content:

```
::marker {
    content: counter(list-item) '□□';
}
```

{% Codepen { user: 'web-dot-dev', id: 'zYWLjBa', height: 500, theme: 'light', tab: 'result' } %}

Our counters automatically increment by one, but we can permit them to increment by a different value if we choose, by setting the `counter-increment` property on the list item. For example, this will increment our counters by three each time:

```
li {
    counter-increment: list-item 3;
}
```

There's a lot more we could delve into with counters. The article CSS Lists, Markers and Counters explains some of the possibilities in greater detail.

Limitations of ::marker styling

There are times when we might want more control over the position and style of our markers. It's not possible to position the marker using flexbox or grid, for example, which can sometimes be a drawback if you require some other alignment. `::marker` exposes a limited number of CSS properties for styling. If our design requires anything other than basic styling, we might be better off using another pseudo-element.

Styling lists that don't look like lists

Sometimes we might want to style our lists in a way that is totally different from the default styling. This is often the case with a navigation menu, for example, where we usually want to remove all markers, and might display our list horizontally, using flexbox. A common practice is to set the `list-style` property to `none`. This will mean the marker pseudo-element will no longer be accessible in the DOM.

{% Aside 'caution' %} A word of warning: `list-style: none` will cause Safari to no longer recognise an unordered list in the accessibility tree—meaning that a screen reader would no longer announce helpful information such as the number of items. A simple fix (assuming your list should be read as such) is to use `role="list"` on the `<ul>` or `<ol>` element in your HTML, as recommended by accessibility expert Scott O'Hara. {% endAside %}

Custom markers with ::before

Styling the `::before` pseudo-element was a common way to create custom list markers before `::marker` came along. But even now, it can allow us more flexibility, when we need it, for visually complex list styling.

Like `::marker`, we can add our own custom bullet styles using the `content` attribute. Unlike using `::marker`, we need to do some manual positioning, as we don't get the automatic benefits offered by `list-style-position`. But we can position it relatively easily with flexbox, and it does open up a greater number of possibilities for alignment. For instance, we could alternate the position of the marker:

{% Codepen { user: 'web-dot-dev', id: 'OJvwZRM', height: 500, theme: 'light', tab: 'result' } %}

If we're styling an ordered list using the `::before` element, we might also wish to use counters to add our numerical markers.

```
li::before {
  counter-increment: list-item;
  content: counter(list-item);
}
```

{% Codepen { user: 'web-dot-dev', id: 'NWYBMRb', height: 500, theme: 'light', tab: 'result' } %}

Using `::before` instead of `::marker` allows us full access to CSS properties for custom styling, as well as permitting animations and transitions—for which support is limited for `::marker`.

List attributes

Ordered lists elements accept some optional attributes, which can help us in a variety of use cases.

Reversed lists

If we have a list of the top 10 albums of the past year, we might want to count down from 10 to 1. We *could* use custom counters for that, and increment them negatively. Or we could simply use the `reversed` attribute in the HTML. I would argue that it generally makes semantic sense to use the `reversed` attribute rather than negatively incrementing the counter in CSS, unless the counters are purely presentational. If the CSS failed to load, you would still see the numbers counting down correctly in the HTML. Plus we need to consider how a screen reader would interpret the list.

Take this demo of the top 10 albums from 2021. If the counters were incremented purely with CSS, someone accessing the page using a screen reader might conclude that the numbers counted upwards, so that number 10 was actually number one.

{% Codepen { user: 'web-dot-dev', id: 'qBoyYaL', height: 500, theme: 'light', tab: 'result' } %}

You can see in the demo that by using the `reversed` attribute, our markers already have the correct value, without any extra effort on our part! But if we're creating custom list markers using the `::before` pseudo-element, we need to adjust our counters. We just need to instruct our list-item counter to increment negatively:

```
li::before {
  counter-increment: list-item -1;
  content: counter(list-item);
}
```

This will be sufficient in Firefox, but in Chrome and Safari the markers will count down from zero to -10. We can fix that by adding the `start` attribute to the list.

{% Codepen { user: 'web-dot-dev', id: 'QWmBrGp', height: 500, theme: 'light', tab: 'result' } %}

Split lists

The `start` attribute allows us to specify the numeric value the list should start from. One way this can be useful is in cases where you want to split a list into groups.

Let's build on our top 10 albums example. Perhaps we actually want to count down the top 20 albums, but in groups of 10. In between the two groups there is some other page content.

{% Img src="image/kheDArv5csY6rvQUJDbWRscckLr1/4hyszzTz5iao3h4SKpmT.png", alt="A wireframed list in columns with an element spanning the columns midway.", width="800", height="776" %}

We'll need to create two separate lists in our HTML, but how can we ensure that the counters will be correct? As our markup currently stands, both lists will count down from 10 to 1, which is not what we want. However, in our HTML we can specify a `start` attribute value. If we add a `start` value of 20 to our first list, the markers will once again be updated automatically!

```
<ol reversed start="20">
  <li>...</li>
  <li>...</li>
  <li>...</li>
</ol>
```

{% Codepen { user: 'web-dot-dev', id: 'JjLBvbw', height: 500, theme: 'light', tab: 'result' } %}

{% Codepen { user: 'web-dot-dev', id: 'YzajLpm', height: 500, theme: 'light', tab: 'result' } %}

Multi-column list layout

Multi-column layout can sometimes lend itself well to our lists, as you can see from the previous demos. By setting a column width, we can ensure our list is automatically responsive, laying itself over two or more columns only when there is sufficient space. We can also set a gap between the columns, and for additional flourish, add a styled column-rule (using a shorthand similar to the `border` property):

```
ol {
    columns: 25rem;
    column-gap: 7rem;
    column-rule: 4px dotted turquoise;
}
```

Using columns, we can sometimes end up with unsightly breaks in our list items—not always the effect we want.

{% Img src="image/kheDArv5csY6rvQUJDbWRscckLr1/bKGN5c3qaE4dnVUYOORr.png", alt="A demonstration of how the content is split between two columns.", width="800", height="325" %}

We can prevent these forced breaks using `break-inside: avoid` on our list items:

```
li {
    break-inside: avoid;
}
```

Custom properties

CSS custom properties open up a whole range of possibilities for styling lists. If we know the index of the list item, we can use it to calculate property values. Unfortunately at present there isn't a way to determine the element's index (in a usable way, at any rate) in CSS alone. Counters only permit us to use their value in the `content` property, and don't allow calculations.

But we *can* set the element's index within the `style` attribute in our HTML, which can make calculations more feasible, especially if we're using a templating language. This example shows how we would set that using Nunjucks:

{% raw %}

```
<ol style="--length: items|length">
  {% for item in items %}
  <li style="--i: {{ loop.index }}">...</li>
  {% endfor %}
</ol>
```

{% endraw %}

Splitting.js is a library that performs a similar function on the client side.

Using the custom property value we can show progression through a list in various ways. One way could be a progress bar for a list of steps. In this example, we're using a pseudo-element with a linear gradient to create a bar for each item that shows how far through the list the user is.

```
li::before {
    --stop: calc(100% / var(--length) * var(--i));
    --color1: deeppink;
    --color2: pink;

    content: '';
    background: linear-gradient(to right, var(--color1) var(--stop), var(--color2) 0);
}
```

{% Codepen { user: 'web-dot-dev', id: 'BarPxpz', height: 500, theme: 'light', tab: 'result' } %}

We could also adjust the hue as the list progresses, by using the `hsl()` color function. We can calculate the `hue` value using our custom property.

{% Codepen { user: 'web-dot-dev', id: 'jOzpxyw', height: 500, theme: 'light', tab: 'result' } %}

Description list styling

As mentioned earlier, we can choose to wrap terms and their definitions within a `div` in a `dl`, to give us more styling options. For example, we might want to display our list as a grid. Setting `display: grid` on the list without a wrapper `div` around each group would mean our terms and descriptions are placed in different grid cells. Sometimes this is useful, as in the following example, showing a menu of pies with their descriptions.

We can define a grid on the list itself and ensure that the terms and descriptions will always align in columns, with the column width determined by the longest term.

{% Codepen { user: 'web-dot-dev', id: 'vYRajmB', height: 500, theme: 'light', tab: 'result' } %}

On the other hand, if we want to distinctly group terms with their descriptions card-style, a wrapper `<div>` is very helpful.

{% Codepen { user: 'web-dot-dev', id: 'OJvwZmN', height: 500, theme: 'light', tab: 'result' } %}

Resources

- Lists An introduction to lists and ::marker
- Custom markers using ::marker
- CSS lists with counters

□□□ #2
□ □□□ □□□ 2026-10-31 21:46:25 UTC
□ □□□ □□□ 2026-10-31 21:46:27 UTC